PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 12/00

A2

(11) International Publication Number: WO 99/44137

(43) International Publication Date: 2 September 1999 (02.09.99)

(21) International Application Number: PCT/US99/03389

(22) International Filing Date: 18 February 1999 (18.02.99)

(30) Priority Data:

60/076,048	26 February 1998 (26.02.98)	US
09/044,915	20 March 1998 (20.03.98)	US

(71) Applicant: SUN MICROSYSTEMS, INC. [US/US]; 901 San Antonio Road, MS UPAL01-521, Palo Alto, CA 94303 (US).

(72) Inventors: SCHEIFLER, Robert; 96 North Street, Somerville, MA 02144 (US). GONG, Li; 917 Florence Lane, Menlo Park, CA 94025 (US).

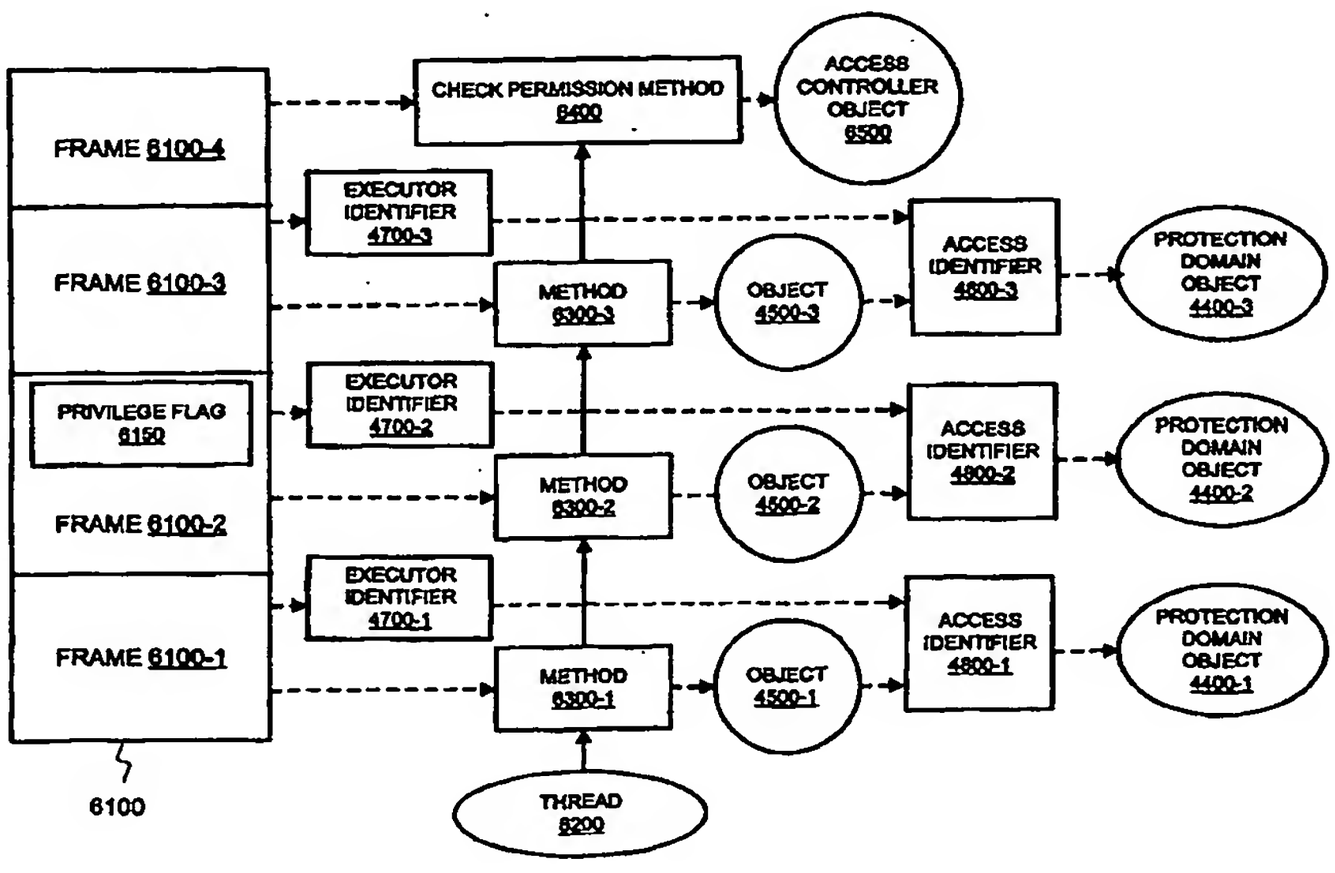
(74) Agents: GARRETT, Arthur, S.; Finnegan, Henderson, Farrabow, Garrett & Dunner, L.L.P., 1300 I Street, N.W., Washington, DC 20005-3315 (US) et al.

(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published

Without international search report and to be republished upon receipt of that report.

(54) Title: STACK-BASED ACCESS CONTROL



(57) Abstract

A system regulates access to resources requested by an operation executing on a computer. The operation invokes a plurality of methods that operate upon code during execution. The system includes a policy file, a call stack, and an execution unit. The policy file stores permissions for each of the resources. The permissions authorize particular types of access to the resource based on a source of the code and an executor of the code. The call stack stores representations of the methods and executors in an order of invocation by the operation. The execution unit grants access to the resource when the types of access authorized by the permissions of all of the methods and executors on the call stack encompass the access requested by the operation.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

STACK-BASED ACCESS CONTROL

RELATED APPLICATIONS

This application is a continuation-in-part of U.S. patent application entitled "Controlling Access to a Resource," filed on December 11, 1997, and accorded Serial No. 08/988,431, which is hereby incorporated by reference.

The following identified U.S. patent applications are relied upon and are incorporated by reference in this application.

U.S. patent application entitled "Protection Domains to Provide Security in a Computer System," filed on December 11, 1997, and accorded Serial No. _____.

U.S. patent application entitled "Secure Class Resolution, Loading and Definition," filed on December 11, 1997, and accorded Serial No. _____.

U.S. patent application entitled "Typed, Parameterized, and Extensible Access Control Permissions," filed on December 11, 1997, and accorded Serial No. _____.

U.S. patent application entitled "Layer-Independent Security for Communication Channels," filed on June 26, 1997, and accorded Serial No. 08/883,636.

Provisional U.S. Patent Application No. 60/076,048, entitled "Distributed Computing System," filed on February 26, 1998.

U.S. Patent Application No. 09/044,923, entitled "Method and System for Leasing Storage," filed on March 20, 1998.

U.S. Patent Application No. 09/044,838, entitled "Method, Apparatus, and Product for Leasing of Delegation Certificates in a Distributed System," filed on March 20, 1998.

U.S. Patent Application No. 09/044,834, entitled "Method, Apparatus and Product for Leasing of Group Membership in a Distributed System," filed on March 20, 1998.

U.S. Patent Application No. 09/044,916, entitled "Leasing for Failure Detection," filed on March 20, 1998.

U.S. Patent Application No. 09/044,933, entitled "Method for Transporting Behavior in Event Based System," filed on March 20, 1998.

U.S. Patent Application No. 09/044,919, entitled "Deferred Reconstruction of Objects and Remote Loading for Event Notification in a Distributed System," filed on March 20, 1998.

U.S. Patent Application No. 09/044,938, entitled "Methods and Apparatus for Remote Method Invocation," filed on March 20, 1998.

U.S. Patent Application No. 09/045,652, entitled "Method and System for Deterministic Hashes to Identify Remote Methods," filed on March 20, 1998.

U.S. Patent Application No. 09/044,790, entitled "Method and Apparatus for Determining Status of Remote Objects in a Distributed System," filed on March 20, 1998.

U.S. Patent Application No. 09/044,930, entitled "Downloadable Smart Proxies for Performing Processing Associated with a Remote Procedure Call in a Distributed System," filed on March 20, 1998.

U.S. Patent Application No. 09/044,917, entitled "Suspension and Continuation of Remote Methods," filed on March 20, 1998.

U.S. Patent Application No. 09/044,835, entitled "Method and System for Multi-Entry and Multi-Template Matching in a Database," filed on March 20, 1998.

U.S. Patent Application No. 09/044,839, entitled "Method and System for In-Place Modifications in a Database," filed on March 20, 1998.

U.S. Patent Application No. 09/044,945, entitled "Method and System for Typesafe Attribute Matching in a Database," filed on March 20, 1998.

U.S. Patent Application No. 09/044,931, entitled "Dynamic Lookup Service in a Distributed System," filed on March 20, 1998.

U.S. Patent Application No. 09/044,939, entitled "Apparatus and Method for Providing Downloadable Code for Use in Communicating with a Device in a Distributed System," filed on March 20, 1998.

U.S. Patent Application No. 09/044,826, entitled "Method and System for Facilitating Access to a Lookup Service," filed on March 20, 1998.

U.S. Patent Application No. 09/044,932, entitled "Apparatus and Method for Dynamically Verifying Information in a Distributed System," filed on March 20, 1998.

U.S. Patent Application No. 09/030,840, entitled "Method and Apparatus for Dynamic Distributed Computing Over a Network," and filed on February 26, 1998.

U.S. Patent Application No. 09/044,936, entitled "An Interactive Design Tool for Persistent Shared Memory Spaces," filed on March 20, 1998.

U.S. Patent Application No. 09/044,934, entitled "Polymorphic Token-Based Control," filed on March 20, 1998.

U.S. Patent Application No. 09/044,944, entitled "Stack-Based Security Requirements," filed on March 20, 1998.

U.S. Patent Application No. 09/044,837, entitled "Per-Method Designation of Security Requirements," filed on March 20, 1998.

BACKGROUND OF THE INVENTION

The present invention is directed to security measures in a computer system and, more particularly, to systems and methods that control access to a resource based on the source of the code and the identity of the principal on whose behalf the code is being executed.

As the use of computer systems grows, organizations are becoming increasingly reliant upon them. A malfunction in the computer system can severely hamper the operation of such organizations. Thus, organizations that use computer systems are vulnerable to users who may intentionally or unintentionally cause the computer system to malfunction.

One way to compromise the security of a computer system is to cause the computer system to execute software that performs harmful actions on the computer system. There are various types of security measures that may be used to prevent a computer system from executing harmful software. One example is to check all software executed by the computer system with a "virus" checker. However, virus checkers only search for very specific software instructions. Therefore, many software-tampering mechanisms go undetected by a virus checker.

Another very common measure used to prevent the execution of software that tampers with a computer's resources is the "trusted developers approach." According to the trusted developers approach, system administrators limit the software that a computer system can access to only software developed by trusted software

developers. Such trusted developers may include, for example, well known vendors or in-house developers.

Fundamental to the trusted developers approach is the idea that computer programs are created by developers, and that some developers can be trusted to produce software that does not compromise security. Also fundamental to the trusted developers approach is the notion that a computer system executes only programs that are stored at locations that are under control of the system administrators.

Recently developed methods of running applications involve the automatic and immediate execution of software code loaded from remote sources over a network. When the network includes remote sources that are outside the control of system administrators, the trusted developers approach does not work.

One conventional attempt to adapt the trusted developers approach to systems that can execute code from remote sources is referred to as the trusted source approach. An important concept of the trusted source approach is the notion that the location from which a program is received (*i.e.*, the "source" of the program) identifies the developer of the program. Consequently, the source of the program may be used to determine whether the program is from a trusted developer. If the source is associated with a trusted developer, then the source is considered to be a "trusted source" and execution of the code is allowed.

One implementation of the trusted source approach is referred to as the sand box method. The sand box method allows all code to be executed, but places restrictions on remote code. Specifically, the sand box method permits all trusted code full access to a computer system's resources and all remote code limited access to the resources. Trusted code is usually stored locally on the computer system under the direct control of the owners or administrators of the computer system, who are accountable for the security of the trusted code.

One drawback of the sand box approach is that the approach is not very flexible because it restricts access by remote code to the same limited set of resources. Conflicts can then arise when remote code from several sources attempt to access the same resources. As a result, conventional systems often limit access by remote code from one source to one set of computer resources, while limiting access by remote

code from another source to a different set of computer resources. For example, a system may limit access by remote code loaded over a network from a source associated with a first computer to one set of files, and similarly limit access by remote code loaded over the network from a source associated with a second computer to another set of files.

Providing security measures that allow more flexibility than the sand box method involves establishing a complex set of relationships between principals and permissions. A "principal" is an entity in the computer system to which permissions are granted. Examples of principals include users, organizations, processes, objects, and threads. A "permission" is an authorization by the computer system that allows a principal to perform a particular action or function.

The task of assigning permissions to principals is complicated by the fact that sophisticated processes may involve the interaction of code from multiple sources. For example, code from a trusted first source being executed by a principal (*e.g.*, a thread) may cause the execution of code from a trusted second source, and then cause execution of code from an untrusted third source.

Even though the principal remains the same when the code from the trusted second source and code from the untrusted third source are executed, the access privileges appropriate for the principal when code from the trusted second source is executed likely differ from access privileges appropriate for the principal when the code from the untrusted third source is being executed. Thus, access privileges appropriate for a principal may change dynamically as the source of the code being executed by the principal changes.

Access privileges may also change dynamically as the principal on whose behalf the code is being executed changes. Sometimes one principal executes code on behalf of another principal. For example, when a principal on one computer requests access to a resource on a remote computer, the request causes a "remote" principal to be invoked on the remote computer to handle the request. Handling of the request by the remote principal may involve the execution of code from trusted and untrusted sources. In these situations, conventional systems continue to base code access privileges on the source of the code without regard to the principal on whose behalf

the code is executed. By failing to consider the principal on whose behalf the code is being executed, conventional systems ignore a possible breach in security.

Based on the foregoing, it is clearly desirable to develop a security mechanism that determines the appropriate code access privileges.

SUMMARY OF THE INVENTION

Systems and methods consistent with the principles of the present invention address this need by determining access control to code based on the source of the code and the principal on whose behalf the code is being executed. By regulating code access based on either or both of these factors, the security in computer systems can be enhanced.

A system consistent with the principles of the present invention regulates access to resources requested by an operation executing on a computer. The operation invokes a plurality of methods that operate upon code during execution. The system includes a policy file, a call stack, and an execution unit. The policy file stores permissions for the resource. The permissions authorize particular types of access to the resource based on a source of the code and an executor of the code. The call stack stores representations of the methods and executors in an order of invocation by the operation. The execution unit grants access to the resource when the types of access authorized by the permissions of all of the methods and executors on the call stack encompass the access requested by the operation.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated in and constitute a part of this specification, illustrate an embodiment of the invention and, together with the description, explain the objects, advantages, and principles of the invention. In the drawings:

Fig. 1 is a diagram of a computer network consistent with the principles of the present invention;

Fig. 2 is a diagram of a computer of Fig. 1 in an implementation consistent with the principles of the present invention;

Fig. 3 is a diagram of a code stream executing in the computer of Fig. 2;

Fig. 4 is a diagram of an exemplary security mechanism illustrating the use of protection domains;

Fig. 5 is a diagram of an exemplary policy implemented through use of the policy file of Fig. 4;

Fig. 6 is a diagram of a call stack associated with a thread executing on the computer of Fig. 2; and

Fig. 7 is a flowchart of processing performed by the check permission method of Fig. 6 in an implementation consistent with the principles of the present invention.

DETAILED DESCRIPTION

The following detailed description of the invention refers to the accompanying drawings. The same reference numbers in different drawings identify the same or similar elements. Also, the following detailed description does not limit the invention. Instead, the scope of the invention is defined by the appended claims.

Systems and methods consistent with the principles of the present invention increase security by providing flexible designation of access privileges to code. The systems and methods not only base the access privileges on the source of the code (*i.e.*, whether the code is trusted or untrusted), but also on the identity of the principal on whose behalf the code is being executed (*i.e.*, whether the principal requesting the code execution is trusted or untrusted).

OVERVIEW OF THE DISTRIBUTED SYSTEM

Methods and systems consistent with the present invention operate in a distributed system ("the exemplary distributed system") with various components, including both hardware and software. The exemplary distributed system (1) allows users of the system to share services and resources over a network of many devices; (2) provides programmers with tools and programming patterns that allow development of robust, secured distributed systems; and (3) simplifies the task of administering the distributed system. To accomplish these goals, the exemplary distributed system utilizes the Java™ programming environment to allow both code and data to be moved from device to device in a seamless manner. Accordingly, the exemplary distributed system is layered on top of the Java programming environment and exploits the characteristics of this environment, including the security offered by

it and the strong typing provided by it. The Java programming environment is more clearly described in Jaworski, Java 1.1 Developer's Guide, Sams.net, 1997, which is incorporated herein by reference.

In the exemplary distributed system, different computers and devices are federated into what appears to the user to be a single system. By appearing as a single system, the exemplary distributed system provides the simplicity of access and the power of sharing that can be provided by a single system without giving up the flexibility and personalized response of a personal computer or workstation. The exemplary distributed system may contain thousands of devices operated by users who are geographically disperse, but who agree on basic notions of trust, administration, and policy.

Within the exemplary distributed system are various logical groupings of services provided by one or more devices, and each such logical grouping is known as a Djinn. A "service" refers to a resource, data, or functionality that can be accessed by a user, program, device, or another service and that can be computational, storage related, communication related, or related to providing access to another user. Examples of services provided as part of a Djinn include devices, such as printers, displays, and disks; software, such as applications or utilities; information, such as databases and files; and users of the system.

Both users and devices may join a Djinn. When joining a Djinn, the user or device adds zero or more services to the Djinn and may access, subject to security constraints, any one of the services it contains. Thus, devices and users federate into a Djinn to share access to its services. The services of the Djinn appear programmatically as objects of the Java programming environment, which may include other objects, software components written in different programming languages, or hardware devices. A service has an interface defining the operations that can be requested of that service, and the type of the service determines the interfaces that make up that service.

Fig. 1 depicts the exemplary distributed system 1000 containing a computer 1100, a computer 1200, and a device 1300 interconnected by a network 1400. The computers 1100 and 1200 may include any conventional computers, such as IBM-

compatible computers, or even "dumb" terminals. During typical operation, computers 1100 and 1200 may establish a client-server relationship to transmit and retrieve data.

The device 1300 may be any of a number of devices, such as a printer, fax machine, storage device, computer, or other devices. The network 1400 may be a local area network, wide area network, or the Internet. Although only two computers and one device are depicted as comprising the exemplary distributed system 1000, one skilled in the art will appreciate that the exemplary distributed system 1000 may include additional computers or devices.

Fig. 2 depicts the computer 1100 in greater detail to show a number of the software components of the exemplary distributed system 1000. One skilled in the art will appreciate that computer 1200 or device 1300 may be similarly configured. Computer 1100 includes a memory 2100, a secondary storage device 2200, a central processing unit (CPU) 2300, an input device 2400, and a video display 2500. The memory 2100 includes a lookup service 2110, a discovery server 2120, and a Java™ runtime system 2130. The Java runtime system 2130 includes the Java™ remote method invocation system (RMI) 2140 and a Java™ virtual machine (JVM) 2150. The secondary storage device 2200 includes a Java™ space 2210.

As mentioned above, the exemplary distributed system 1000 is based on the Java programming environment and thus makes use of the Java runtime system 2130. The Java runtime system 2130 includes the Java™ application programming interface (API), allowing programs running on top of the Java runtime system to access, in a platform-independent manner, various system functions, including windowing capabilities and networking capabilities of the host operating system. Since the Java API provides a single common API across all operating systems to which the Java runtime system 2130 is ported, the programs running on top of a Java runtime system run in a platform-independent manner, regardless of the operating system or hardware configuration of the host platform. The Java runtime system 2130 is provided as part of the Java™ software development kit available from Sun Microsystems of Mountain View, California.

The JVM 2150 also facilitates platform independence. The JVM 2150 acts like an abstract computing machine, receiving instructions from programs in the form of byte codes and interpreting these byte codes by dynamically converting them into a form for execution, such as object code, and executing them. RMI 2140 facilitates remote method invocation by allowing objects executing on one computer or device to invoke methods of an object on another computer or device. The RMI may be located within the JVM, and both the RMI and the JVM are provided as part of the Java software development kit.

The lookup service 2110 defines the services that are available for a particular Djinn. That is, there may be more than one Djinn and, consequently, more than one lookup service within the exemplary distributed system 1000. The lookup service 2110 contains one object for each service within the Djinn, and each object contains various methods that facilitate access to the corresponding service. The lookup service 2110 and its access are described in greater detail in co-pending U.S. Patent Application No. 09/044,826, entitled "Method and System for Facilitating Access to a Lookup Service," which has previously been incorporated by reference.

The discovery server 2120 detects when a new device is added to the exemplary distributed system 1000 during a process known as boot and join or discovery, and when such a new device is detected, the discovery server passes a reference to the lookup service 2110 to the new device, so that the new device may register its services with the lookup service and become a member of the Djinn. After registration, the new device becomes a member of the Djinn, and as a result, it may access all the services contained in the lookup service 2110. The process of boot and join is described in greater detail in co-pending U.S. Patent Application No. 09/044,939, entitled "Apparatus and Method for providing Downloadable Code for Use in Communicating with a Device in a Distributed System," which has previously been incorporated by reference.

The Java space 2210 is an object repository used by programs within the exemplary distributed system 1000 to store objects. Programs use the Java space 2210 to store objects persistently as well as to make them accessible to other devices

within the exemplary distributed system. Java spaces are described in greater detail in co-pending U.S. Patent Application No. 08/971,529, entitled "Database System Employing Polymorphic Entry and Entry Matching," assigned to a common assignee, filed on November 17, 1997, which is incorporated herein by reference. One skilled in the art will appreciate that the exemplary distributed system 1000 may contain many lookup services, discovery servers, and Java spaces.

FUNCTIONAL OVERVIEW

A security enforcement mechanism is provided in which the access permissions of a thread are allowed to vary over time based on the source and executor of the code currently being executed. The source of the code indicates whether the code is from a trusted or untrusted source. The executor indicates the principal on whose behalf the code is being executed. For example, the executor may be a particular user or a particular organization on whose behalf the process or program is operating on a client computer.

When a routine that arrives from a trusted source is executing, the thread executing the routine is typically allowed greater access to resources. Similarly, a trusted executor may be given greater access to resources.

When a routine calls another routine, the thread executing the routines is associated with permissions common to both routines. Thus, the thread is restricted to a level of access that is less than or equal to the level of access allowed for either routine.

The mechanism allows certain routines to be "privileged." When determining whether a thread is able to perform an action, only the permissions associated with the privileged routine and the routines above the privileged routine in the calling hierarchy of the thread are inspected.

According to an implementation consistent with the present invention, the security mechanism described herein uses permission objects and protection domain objects to store information that models the security policy of a system. The nature and use of these objects, as well as the techniques for dynamically determining the time-variant access privileges of a thread, are described hereafter in greater detail.

TRUSTED AND UNTRUSTED SOURCES

Fig. 3 is a diagram of a code stream 3100 executing in computer 1100 (Fig. 2). The code stream 3100 is executed by a code execution element 3200, such as JVM 2150, and may derive from zero or more untrusted sources 3300 or zero or more trusted sources 3400. Untrusted sources 3300 and trusted sources 3400 may be file servers, including file servers connected to the Internet, or other similar devices. An untrusted source is typically not under the direct control of the operator of computer 1100. Code from untrusted sources is herein referred to as untrusted code.

Because untrusted code is considered to pose a high security risk, the set of computer resources that untrusted code may access is usually restricted to those which do not pose security threats. Code from a trusted source is code usually developed by trusted developers. Trusted code is considered to be reliable and poses much less security risk than untrusted code.

Software code which is loaded over the network from a remote source and immediately executed is herein referred to as remote code. Typically, a remote source is a computer system of a separate organization or individual. The remote source is often connected to the Internet.

Normally untrusted code is remote code. However, code from sources local to computer 1100 may pose a high security risk. Code from such local sources may be deemed to be untrusted code from an untrusted source. Likewise, code from a particular remote source may be considered to be reliable and to pose relatively little risk, and thus may be deemed to be trusted code from a trusted resource.

According to an implementation consistent with the present invention, a security mechanism is used to implement security policies that allow trusted code to access more resources than untrusted code, even when the trusted and untrusted code are executed by the same principal. A security policy determines what actions code execution element 3200 will allow the code within code stream 3100 to perform. The use of permissions and protection domains allows policies that go beyond a simple trusted/untrusted dichotomy by allowing relatively complex permission groupings and relationships.

Protection domains and policies that may be used in conjunction with typed permissions will be described in greater detail with reference to Fig. 4.

TRUSTED AND UNTRUSTED EXECUTORS

The user or organization on whose behalf a computer program is operating (or in some circumstances, the program itself) is known as the "executor" (*i.e.*, the principal on whose behalf resources will be accessed). The executor for a program on the computer 1200 (a "client executor"), for example, may be different than the executor for a program on the computer 1100 (a "server executor")

Code execution element 3200 receives the request on behalf of a client executor via the RMI 2140 (Fig. 2). In response, code execution element 3200 executes an operation, such as a thread, to handle the request. The thread is responsible for obtaining the appropriate code and/or resources to satisfy the request, and the thread will, in general, be permitted to operate on behalf of either or both of the server executor and the client executor.

Code execution element 3200 permits authorized executors, or "trusted executors," greater access to computer resources because the trusted executors are not considered to pose a high security risk. Trusted executors may include system operators that need greater access to the computer resources to handle system updates and the like. Unauthorized executors, or "untrusted executors," are treated differently. Untrusted executors are considered to pose a high security risk and, therefore, are given limited access to the computer resources.

According to an implementation consistent with the present invention, a security mechanism is used to implement security policies that allow trusted executors to access more resources than untrusted executors, even when the trusted and untrusted executors request code from a single source. A security policy determines what actions code execution element 3200 will allow. The use of permissions and protection domains allows policies that go beyond a simple trusted/untrusted dichotomy by allowing relatively complex permission groupings and relationships.

Protection domains and policies that may be used in conjunction with typed permissions shall now be described in greater detail with reference to Fig. 4.

EXEMPLARY SECURITY MECHANISM

An exemplary security mechanism illustrating the use of protection domains is shown in Fig. 4. The exemplary security mechanism includes a policy file 4100, a policy object 4200, a domain mapper object 4300, and one or more protection domain objects 4400. The security mechanism is implemented using the code execution element 3200 (Fig. 3).

Code execution element 3200 executes the code it receives from code stream 3100 (Fig. 3). For the purpose of explanation, it shall be assumed that the code from code stream 3100 is object-oriented software. Consequently, the code is in the form of methods associated with objects that belong to classes. In response to instructions embodied by code executed by code execution element 3200, code execution element 3200 creates one or more objects 4500. An object is a data structure containing data combined with the procedures or functions that manipulate the data. All objects belong to a class, such as class 4600. Each object belonging to a class has the same fields ("attributes") and the same methods. The methods are the procedures, functions, or routines used to manipulate the object. An object is said to be an "instance" of the class to which the object belongs.

One or more class definitions are contained in the code from code stream 3100. The fields and methods of the objects belonging to a class are defined by a class definition. These class definitions are used by code execution element 3200 to create objects which are instances of the classes defined by the class definitions.

These class definitions are generated from source code written by a programmer. For example, a programmer using a Java development kit enters source code that conforms to the Java programming language into a source file. The source code embodies class definitions and other instructions which are used to generate byte codes that control the execution of the code execution element 3200. Techniques for defining classes and generating code executed by a code execution element, such as a Java virtual machine, are well known to those skilled in the art.

Each class defined by a class definition from code stream 3100 is associated with a class name 4620 and a code identifier 4640. Code execution element 3200

maintains an association between a class 4600 and its class name 4620 and code identifier 4640. The code identifier 4640 represents a source of the code.

A "source of code" is an entity from which computer instructions are received. Examples of sources of code include a file or persistent object stored on a data server connected over a network, a Flash EPROM reader that reads instructions stored on a Flash EPROM, or a set of system libraries.

In an implementation consistent with the present invention, the code identifier 4640 is a composite record containing a uniform resource locator ("URL") 4642 and a set of public cryptographic keys 4644. A URL identifies a particular source. The URL 4642 is a string used to uniquely identify any server connected to the Internet. The URL 4642 may also be used to designate sources local to computer 1100. Typically, the URL 4642 includes a designation of the file and the directory of the file that is the source of the code stream that a server is providing.

A public cryptographic key, herein referred to as a "key," is used to validate the digital signature which may be included in a file used to transport related code and data. Public cryptographic keys and digital signatures are described in further detail in Schneier, Applied Cryptography, 1996. The keys 4644 may be contained in the file, contained in a database associating keys with sources (*e.g.*, URLs), or accessible using alternative techniques.

A class may be associated with the digital signature included in the file used to transport code defining the class, or the class definition of the class may be specifically associated with a digital signature. A class that is associated with a valid digital signature is referred to as being signed. Valid digital signatures are digital signatures that can be verified by known keys stored in a database. If a class is associated with a digital signature that cannot be verified, or the class is not associated with any digital signature, the class is referred to as being unsigned. Unsigned classes may be associated with a default key. A key may be associated with a name that may be used to look up the key in a database.

While one code identifier format has been described as including data indicating a source (*i.e.*, cryptographic key and URL), alternate formats are possible.

Other information indicating the source of the code, or combinations thereof, may be used to represent code identifiers.

An executor identifier 4700 represents the executor of code. An "executor of code" is a principal (*e.g.*, a user or organization) on whose behalf the code is being executed. An example of an executor might include a person, like "John T. Smith," or an organization, like "Sun Microsystems, Inc." An "executor identifier" is, therefore, some form of identifier that represents the executor. Examples of possible executor identifiers include string names, computer system login names, and employee numbers. When a server receives a request from a client via the RMI, the server may require authentication of the client executor as proof that the client program is executing on behalf of the client executor.

PROTECTION DOMAINS AND PERMISSIONS

According to an implementation consistent with the present invention, protection domains are used to enforce security within computer systems. A protection domain can be viewed as a set of permissions granted to one or more executors when code from one or more sources is being executed on their behalf. A permission is an authorization by the computer system that allows a principal to execute a particular action or function. Typically, permissions involve an authorization to perform an access to a computer resource in a particular manner. An example of an authorization is an authorization to "write" to a particular directory in a file system (*e.g.*, /home).

A permission can be represented in numerous ways in a computer system. For example, a data structure containing text instructions can represent permissions. An instruction such as "permission executor write /somedirectory/somefile" denotes a permission to write to file "somefile" in the directory "/somedirectory" on behalf of the principal "executor." The instruction denotes which particular action is authorized, the executor authorized to perform the action, and the computer resource upon which that particular action is authorized. In this example, the particular action authorized is to "write" on behalf of the principal "executor." The computer resources upon which the particular action is authorized is a file "/somedirectory/somefile" in a file system of computer 1100. In the example, the file and the directory in which the

file is contained are expressed in a conventional form recognized by those skilled in the art.

Permissions can also be represented by objects, herein referred to as permission objects. Attributes of the object represent a particular permission. For example, an object can contain an action attribute of "write," and a target resource attribute of "/somedirectory." A permission object may have one or more permission validation methods which determine whether a requested permission is authorized by the particular permission represented by the permission object.

POLICIES

The correlation between permissions, executors, and code sources constitutes the security policy of the system. The policy of the system may be represented by one or more files containing instructions. Each instruction establishes a mapping between a particular access identifier and a particular authorized permission. An access identifier is composed of an executor identifier and a code identifier. The permission specified in an instruction applies to all objects that belong to classes that are associated with the code identifier specified in the access identifier of the instruction, when those objects are operated on behalf of the executor specified by the executor identifier in the access identifier of the instruction.

Fig. 5 illustrates an exemplary policy implemented through use of the policy file 4100 (Fig. 4). The format of an instruction in exemplary policy file 4100 is:

<"permission"> <executor> <URL> <key name> <action> <target>

The <executor> identifies the executor of the code; the combination of the <URL> and the key that corresponds to <key name> constitute a code source; and the <action> and <target> represent a permission. The key is associated with a key name. The key and the corresponding key name are stored together in a key database. The key name can be used to find the key in the key database. For example, consider the following instruction:

permission executor1 file://somesource somekey write /tmp/*

The above instruction represents an authorization of a permission for executor "executor1" to write to any file in "/tmp/*" by an object that belongs to the class associated with the code source "file://somesource" -"somekey" (*i.e.*, URL-key name).

IMPLIED PERMISSIONS

One permission does not have to exactly match another permission to be considered "encompassed" by the other permission. When a first permission encompasses a second permission without matching the second permission, the first permission is said to "imply" the second permission. For example, a permission to write to any file in a directory, such as "c:/," implies a permission to write to any specific file in the directory, such as "c:/thisfile." As another example, a permission to read the file "d:/log" granted to "all current employees of Sun Microsystems, Inc." implies a permission to read the file "d:/log" granted to a specific employee of that same organization.

If a permission is represented by a permission object, the validation method for the permission object contains code for determining whether one permission is implied by another. For example, a permission to write to any file in a directory implies a permission to write to any specific file in that directory, and a permission to read from any file in a directory implies a permission to read from any specific file in that directory. However, a permission to write does not imply a permission to read.

POLICY IMPLEMENTING OBJECTS

A variety of objects may be used to implement the policy represented by the access identifiers to permissions mapping contained in policy file 4100. According to the implementation illustrated in Fig. 4, in order to efficiently and conveniently implement the policy, policy object 4200, domain mapper object 4300, one or more protection domain objects 4400, and one or more access identifiers 4800 are provided.

Policy object 4200 is an object for storing the policy information obtained, for example, from policy file 4100. Specifically, policy object 4200 provides a mapping of access identifiers to permissions, and is constructed based on the instructions within policy file 4100. Within the policy object 4200, the access identifiers and their associated authorized permissions may be represented by data structures or objects.

Protection domain objects 4400 are created on demand when new access identifiers 4800 are encountered by domain mapper object 4300. When an access identifier 4800 is received, domain mapper object 4300 determines whether a protection domain object 4400 is already associated with the access identifier 4800.

The domain mapper object 4300 maintains data indicating which protection domain objects have been created and the access identifiers associated with the protection domain objects. If a protection domain object is already associated with the access identifier, the domain mapper object 4300 adds a mapping of the access identifier and protection domain object to a mapping of access identifiers and protection domain objects maintained by the domain mapper object 4300.

If a protection domain object is not associated with the access identifier, a new protection domain object is created and populated with permissions. The protection domain object is populated with those permissions that are mapped to the access identifier based on the mapping of access identifiers to permissions in the policy object 4200. Finally, the domain mapper object 4300 adds a mapping of the access identifier and protection domain object to the mapping of access identifiers and protection domain objects as previously described.

In other implementations consistent with the present invention, instead of storing the mapping of access identifiers to protection domain objects in a domain mapper object, the mapping is stored as static fields in the protection domain class. The protection domain class is the class to which protection domain objects 4400 belong. There is only one instance of a static field for a class no matter how many objects belong to the class. The data indicating which protection domain objects have been created and the access identifiers associated with the protection domain objects is stored in static fields of the protection domain class.

Static methods are used to access and update the static data mentioned above. Static methods are invoked on behalf of the entire class, and may be invoked without referencing a specific object.

EXEMPLARY CALL STACK

The permission objects, protection domain objects, and policy objects described above are used to determine access rights of a thread. According to an implementation consistent with the present invention, such access rights vary over time based on what code the thread is currently executing, and on which executor's behalf the thread is currently executing. The sequence of calls that resulted in execution of the currently executing code of a thread is reflected in the call stack of

the thread. Reference to an exemplary call stack shall be made to explain the operation of a security mechanism that enforces access rights in a way that allows the rights to vary over time.

Fig. 6 is a block diagram that includes a call stack 6100 associated with a thread 6200 in which the method 6300-1 of an object 4500-1 calls the method 6300-2 of another object 4500-2 that calls the method 6300-3 of yet another object 4500-3 that calls a check permission method 6400 of an access controller object 6500.

Thread 6200 is a thread executing on computer 1100. Call stack 6100 is a stack data structure representing a calling hierarchy of the methods invoked by thread 6200 at any given instance. At the instance illustrated in Fig. 6, call stack 6100 contains a frame (*e.g.*, frame 6100-1) for each method executed by thread 6200, but not yet completed.

Each frame corresponds to the method that has been called but not completed by thread 6200. The relative positions of the frames on the call stack 6100 reflect the invocation order of the methods that correspond to the frames. When a method completes, the frame that corresponds to the method is removed from the top of the call stack 6100. When a method is invoked, a frame corresponding to the method is added to the top of the call stack 6100.

Each frame contains information about the method and the object that correspond to the frame. From this information, the class of the method can be determined by invoking a "get class" method provided for every object by the code execution element 3200. The code identifier of this class can then be determined from the association maintained by the code execution element 3200. Each frame also contains the executor identifier (*e.g.*, executor identifier 4700-1) of the executor on whose behalf the thread is executing. The executor identifier and code identifier can then be composed into an access identifier (*e.g.*, access identifier 4800-1). From the mapping in domain mapper object 4300, the protection domain object associated with the access identifier for a given frame can be determined.

For example, assume thread 6200 invokes method 6300-1. While executing method 6300-1, thread 6200 invokes method 6300-2. While executing method 6300-2, thread 6200 invokes method 6300-3. While executing method 6300-3, thread 6200

invokes method 6400. At this point, call stack 6100 represents the calling hierarchy of methods as shown in Fig. 6. Frame 6100-4 corresponds to method 6400, frame 6100-3 to method 6300-3, frame 6100-2 to method 6300-2, and frame 6100-1 to method 6300-1. When thread 6200 completes method 6400, frame 6100-4 is removed from the call stack 6100.

METHOD/PERMISSION RELATIONSHIPS

Each frame on the call stack 6100 is associated with a set of permissions. The set of permissions for a given frame is determined by the protection domain object associated with the source from which the code for the given method was received and the principal on whose behalf the code is being executed. The relationship between frames, protection domains, and permissions shall now be described with continued reference to Fig. 6.

Protection domain object 4400-1 is mapped from the access identifier 4800-1 formed by the executor identifier 4700-1 and the code identifier of the class of object 4500-1. Method 6300-1 of object 4500-1 invokes method 6300-2 of object 4500-2 on behalf of executor identifier 4700-2. Protection domain object 4400-2 is mapped from the access identifier 4800-2 formed by the executor identifier 4700-2 and the code identifier of the class of object 4500-2. Method 6300-2 of object 4500-2 invokes method 6300-3 of object 4500-3 on behalf of executor identifier 4700-3. Protection domain object 4400-3 is mapped from the access identifier 4800-3 formed by the executor identifier 4700-3 and the code identifier of the class of object 4500-3.

While protection domain objects are used to organize and determine the access rights of a particular executor and code source, some mechanism must be provided to determine the access rights of a thread having a call stack with multiple methods whose code arrived from multiple sources or whose code is requested to be executed on behalf of multiple principals. According to an implementation consistent with the present invention, this determination is performed by an access controller object, as shall be described in greater detail hereafter.

EXEMPLARY ACCESS CONTROLLER

According to an implementation consistent with the present invention, an access controller object is used to determine whether a particular action may be

performed by a thread. Specifically, before a resource management object accesses a resource, the resource management object (*e.g.*, object 6300-3) invokes a check permission method 6400 of an access controller object 6500.

In the illustrated example, the resource management method 6300-3 invokes a check permission method 6400 of the access controller object 6500 to determine whether access to the resource is authorized. To make this determination, the check permission method 6400 of the access controller object 6500 performs the steps that shall be described with reference to Fig. 7.

DETERMINING WHETHER AN ACTION IS AUTHORIZED

According to an implementation consistent with the present invention, an action is authorized if the permission required to perform the action is included in each protection domain object associated with the thread at the time when a request to determine the authorization is made. A permission is said to be included in a protection domain object if that permission is encompassed by one or more permissions associated with the protection domain object. For example, if an action requires permission to write to a file in the "e:/tmp" directory on behalf of the principal "Bob," then that required permission would be included in protection domain object 4400-1 if the protection domain object 4400-1 explicitly contains or implies that permission.

Assume that thread 6200 is executing method 6300-3 when thread 6200 makes a request for a determination of whether an action is authorized by invoking the check permission method 6400. Assume further that thread 6200 has invoked method 6300-1, method 6300-2, and method 6300-3 and these methods have not completed when thread 6200 invoked method 6400. The protection domain objects associated with thread 6200 when the request for a determination of authorization is made are represented by protection domain objects 4400-1, 4400-2, and 4400-3.

Given the calling hierarchy present in the current example, the required permission to perform an action of writing to file "d:/sys/pwd" on behalf of "Bob" is not authorized for thread 6200 because the required permission is not encompassed by any permission included in protection domain object 4400-1, if the only permission contained therein is "write to e:/tmp."

PRIVILEGED METHODS

Sometimes the need arises to authorize an action that a method performs irrespective of the protection domain objects associated with the methods that precede the method in the calling hierarchy of a thread. Updating a password is an example of when such a need arises.

Specifically, because the security of a password file is critical, the permissions required to update the password file are limited to very few specialized protection domain objects. Typically, such protection domain objects are associated with methods of objects from trusted code and trusted executors that provide their own security mechanisms. For example, a method for updating a password may require the old password of a user before updating the new password for that user. The method may also require authentication of the principal on whose behalf the update is being requested, and permit updating of the password for only authorized principals.

Because permissions to update passwords are limited to code from specific sources and to code executed on behalf of specific authorized principals, code from all other sources or principals will not be allowed to update the passwords. This is true even in a situation such as that shown in Fig. 6, where code from a remote source (method 6300-1) attempts to change the password by invoking trusted code (method 6300-3) which has permission to update the password. Access is denied in this situation because at least one method in the calling hierarchy (method 6300-1) does not have the necessary permission.

According to an implementation consistent with the present invention, a privilege mechanism is provided to allow methods that do not themselves have the permission to perform actions to nevertheless cause the actions to be performed by calling special "privileged" methods that do have the permissions. This result is achieved by limiting the protection domain objects that are considered to be "associated with a thread" to only those protection domain objects that are associated with a "privileged" method and those methods that are subsequent to the privileged method in the calling hierarchy.

A method may cause itself to be privileged (*i.e.*, enable the privilege mechanism) by invoking a method of a privilege object called for example,